CSCE 5200 Final Report  
Search Engine

Jordan Sanders

University of North Texas

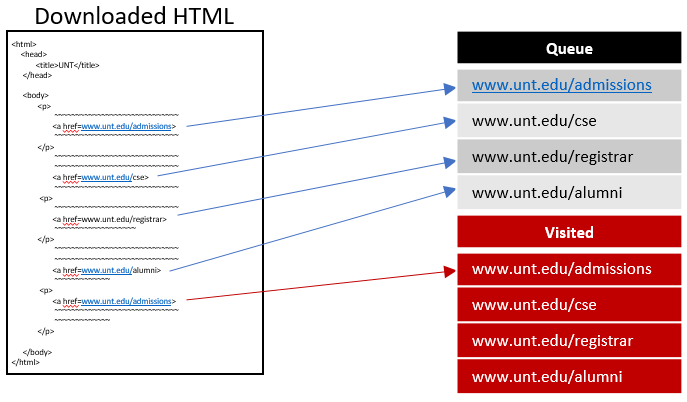
Download Link: https://www.dropbox.com/sh/nlx4bu0j1ulkgrc/AADfJLt9cbOvpPPMb9AWeaD5a?dl=0

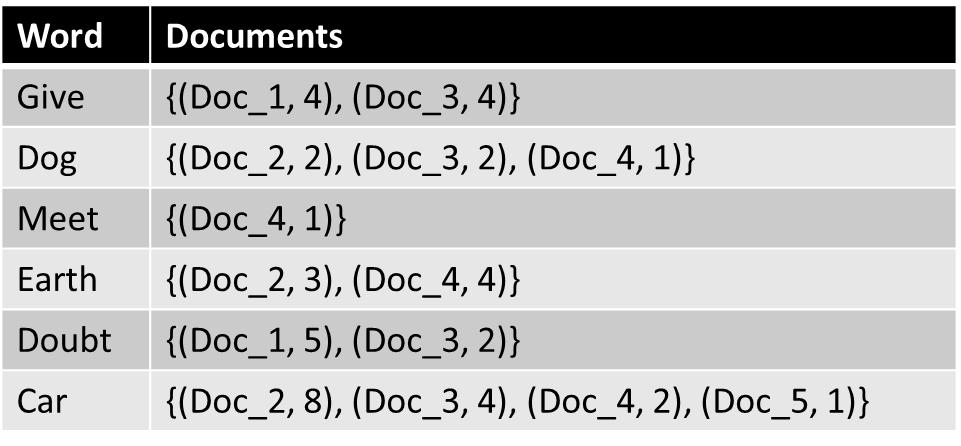
# Abstract

This report covers the implementation of a TF-IDF weighted search engine for querying the domain www.unt.edu. It will also include information about the challenges accompanying the implementation, a discussion of the weighting model compared to other weighting schemes, and also a manual evaluation of the search results for several queries with comparisons to more state of the art search engine results.

# Implementation

This search engine for the UNT domain consists of three primary components. The first component is the web crawler, this gathers the data to be searched. The second component is the creation of a vector space model which houses information about the gathered data. The last component is the actual query code, it uses the model created by the previous component to return a list of web addresses ranked by greatest similarity to the query. All the code is written in Python 2.7 and relies on several key libraries. In the following section I will detail how each of the primary components work.

Gathering the dataset for the search engine is done by the web crawler. The web crawler begins on the domain <https://www.unt.edu> and downloads the html file associated with that address using the library urllib2. The document is then converted to a BeautifulSoup4 object which extracts all tagged elements from the html and makes them available as lists. All of the links are then obtained by grabbing the “href” attribute from each “<a>” tag in the file. Each link is then added to a queue object from the Queue library. Queues are ‘first in first out” structures, by using this the links are followed in a breadth first search manner. After all links are extracted the document is converted to plain text with all html removed using the BeautifulSoup4’s get text function and removing all remaining scripts. The plain text document is then saved in a cPickle binary file as a python tuple with the first element as the web address. Below is an example of the web crawling algorithm.

Using the dataset created by crawling 3000 websites within the <https://www.unt.edu> domain, a model of these documents consisting of weight information can be made. This weight information makes it possible to rank documents by their similarity to the terms present in the query. Weights are calculated in a single pass through every document in the data set. The steps required in this pass are tokenization, stop word removal, word stemming, and TF-IDF calculation. Word tokenization in my project is accomplished using a regex subtraction parse. The regex pattern removes whitespace characters, punctuation, and formatting characters from the document word list. The document word list is then converted to lowercase for stop word removal and the stemming process. Included in my project is a text file containing stop words, which is parsed and added to a list. This list is then compared to all words in the document word list and anything present in both is removed from the document word list. This project uses the Porter Stemmer implemented in the Natural Language Tool Kit library (NLTK). It takes a list of words and out puts a list of all the stemmed versions of those words, using the Porter Stemmer rule set.The last step of the model creation component is to gather and calculate TF-IDF, term frequency and inverse document frequency. Term frequency is gathered first in a dictionary object with the word as a key. The number of unique words in the text is then acquired by creating a set from the document’s word list. The term frequency is then permanently stored in a dictionary of lists where the word is the key and the list contains tuples of the document id and number of occurrences of that word in the document divided by the number of unique words in that document. The inverse document frequency for every word is then calculated and stored in a dictionary where the word is the key. The calculation for each word’s inverse document frequency is a sum of the log of the number of documents in the dataset divided by the term frequency of the word. Below is an example of the term frequency dictionary for each word.

Querying the model of the crawled webpages is the final component of the project. This process takes in a query from the user and preprocesses it using the same functions that the web crawler uses to tokenize and convert to lowercase. The cosine similarity between the query and all documents in the model can then begin. To calculate this, we find all documents containing all the query words using the list of tuples created earlier. Each document is then added as a key to a dictionary where a sum is kept of term frequencies multiplied by the inverse document frequency squared for that word. After moving through each word of the query the square root of every document’s sum is taken. The last step is to order the documents by their similarity in descending order, giving us a list of document similarity pairs. Results are returned to the user by printing the first 5 elements of the list.

# Challenges

In this section I will detail some of the challenges I faced during the implementation of the search engine.

While making the web crawler component of the project, there were four main hurdles that I faced. The first of these was staying on the “unt.edu” domain without moving to outside links. I solved this problem by using a regex that required the pattern “unt.edu” to be inside the crawled link. The second challenge was to only check a domain a single time. For this problem, I created a separate list from the queue of links where an incoming link would be cross checked with, stopping any previously visited links from being visited again and clogging up the queue. Another problem that I solved with regex was downloading only html pages. Urllib2 is not picky about what website it grabs and there are many links in the UNT domain that are not html, such as PowerPoints and mp3 files. I created a regex pattern that would only grab links with no suffix or one of the two html versions of a suffix. The final problem I faced during the creation of this component was thoroughly removing all html tags and leaving only the page’s substance text. For this task, I used BeautifulSoup4’s get text function, however that leaves all scripts inside the text untouched. To remove these scripts, I looped through the BeautifulSoup4 object and removed all script tags left in the returned get text.

In the creation of the search engine model, the challenges were almost entirely related to the preprocessing of the text for inclusion in the model. Preprocessing involved the removal of all punctuation, whitespace, and end line terminators. Subtraction regular expressions were used to solve all of these problems. Using Python’s built in split function, all words were tokenized into a list. Stemming is another preprocessing step that involves shortening words to their base forms. The rules for the Porter Stemming method are very thorough and exhaustive, so to accomplish this task I imported a Stemmer from the Natural Language Tool Kit which takes in lists of words.

An important part of the make model component was to store all the required data for the TF-IDF model in a single pass through of the documents and only two loops through the text. This reduces the time required to create the model at run time. I further reduced this time by using Python’s set function for the inverse document frequency portion so that repeated words wouldn’t require multiple cross referencing.

# Discussion

While this search engine works, it relies on a weak model of query document relationships. This model is unlikely to perform very well on a diverse domain space where more than just a university website is being utilized and is highly susceptible to system gaming. Currently searching for “admissions” would return what I am most likely looking for if I knew I was searching solely in UNT’s system, however even if we only expand this to include several universities, that search return may no longer be exactly what I want, requiring greater specification. Websites being cataloged by this system are also easily able to influence search rankings with a detrimental effect to the end user. If I were a website owner, I could simply hide text on my page that repeats keywords over and over which would increase my ranking similarity.

Cosine similarity used on the TF-IDF model here is somewhat naïve as it looks for only exact word similarity between documents. A document based solely on dogs should be included if I search the database for canine, however TF-IDF would not be able to correlate the two documents in anyway. A better approach for cosine similarity would be to use word embeddings which capture the meaning of a word, enabling the search engine to relate canine and dog to each other. State of the art similarity measures use Word2Vec which creates word vectors relating words to one another. There are even improvement to be made with cosine similarity itself, such as Word Mover’s Distance that uses word vectors and a unique distance calculation that provide increased similarity accuracy.

State of the art search engines such as Google use heuristics to augment search results and improve them. Relying solely on a naïve document similarity measure to rank search results will potentially steer people in the wrong direction. An example is for someone who lives in Dallas to search for a car dealership of a certain brand. If the highest document similarity measure returns a car dealership in a different city, this is not useful for the individual. Heuristics that consider location data, time, previous search history, and other useful information can provide more accurate results to the user.

# Evaluation

The first of my three evaluations is for the term “admissions”. A user querying the UNT domain for this term is most likely looking for the admissions office or how to apply to UNT. My search engine’s top result for this term is a graduate admissions page for the music department at UNT. This page is relevant to the query; however, it is a specific department’s admissions page and most users would not be looking for this page. The remaining four results are all to the UNT primary admissions office, including freshman, transfer, and international student pages. These four results are most likely what a user would be looking for in this case. Querying Google for the same term and limiting it to the UNT domain yields results like these.

My second search term to evaluate the search engine is “dean”, which would most likely be searched for by students seeking the dean of students. Without a specific department included all other deans are not optimal returns for this query. My search engine only includes a single result for the dean of students and it is not the top result. The remaining results are all “people” or contact pages for various departments, including the vice provost office. Googling this query leads to the top two results being the dean of students and the remainder are various department’s main pages.

The last search term I will cover is for “Blanco”, my advisor professor. I typically use Google as a quick way of accessing his personal web page in the UNT domain, most likely what other users would do as well. My search engine returns results containing the HiLT lab, Dr. Nielsen who Dr. Blanco works with, a page recognizing members of the HiLT lab, and also a list of people on an advisory board that Dr. Blanco is not a part of. These results failed to return the most important result, his personal page, however it is likely that his personal page is missing from the search engine’s crawled data. In this instance, the search engine does a decent job of finding results near Dr. Blanco, but fails due to a lack of data. Google on the other hand does not have this problem and returns Dr. Blanco’s personal page as the top result.